Performance Analysis of the 1GHz Motorola G4 RISC processor versus the 1.13 GHz P3 and 2.4 GHz P4 Intel CISC processors

P. Aaron Lott
AMSC 662
Final Report

Abstract

We discuss the implementation and analysis of optimization analysis code that tests Motorola 745x RISC and Intel x86 architectures.

Introduction

To begin my project I started out with the assembly code from the text book [1] that measures the number of cycles it takes to complete a routine on a x86 based machine. I then searched through two technical manuals for the Motorola G4 RISC chip to find out how to access the cycle counter, I was able to find the register where the data is stored namely PMC_1, but nothing was said on how to access it or how often the register is updated. Fortunately I was able to use some frameworks called CHUD developed by Apple's Architecture and Performance Group. With this tool, not only was I able to access the cycle counter, but also the instruction counter, which we found to be important in analyzing the performance of the Chip. On the Intel side, I re-wrote and tested the cycle counter code so that I could measure runs on x86 machines. To test the machines, I re-wrote the vector summation functions "The combine codes" from lecture notes so that I could define arbitrary data types, i.e. int, float, double, or even abstract data types if one had an application they wish to test using such. I then structured my code using the C preprocessor #ifdef statements so that one can simply choose from a parameter file the data type, size of vector, and type of test, i.e. G4 or x86 cycle counters. Finally I created a Makefile to manage compilation flags and compilers. (At this time only gcc has been used successfully.)

Code Rundown

I've kept my code very general so that one will be able to use this to test their own scheme. I developed the user interface so one modifies the parameter and Makefiles file to define a function in which to test. Excerpts from the README file:

```
To implement your own routine you will need to first define the function to
receive values void my_func(vec_ptr v, DATATYPE *dest). (you may want to use
one of the combine* functions as a template).  Then call the function from
the call_combine.c function. You'll be able to insert your code beneath the
code for combine6aaa for example. You will also want to create a variable for
your count. Depending on the architecture you wish to test you'll either
define it in the code block
```

```
#ifdef X86_ON
double count1,count2,count3,count4,count4p,count5,count6,count6aa,count6aaa;
#endif
```

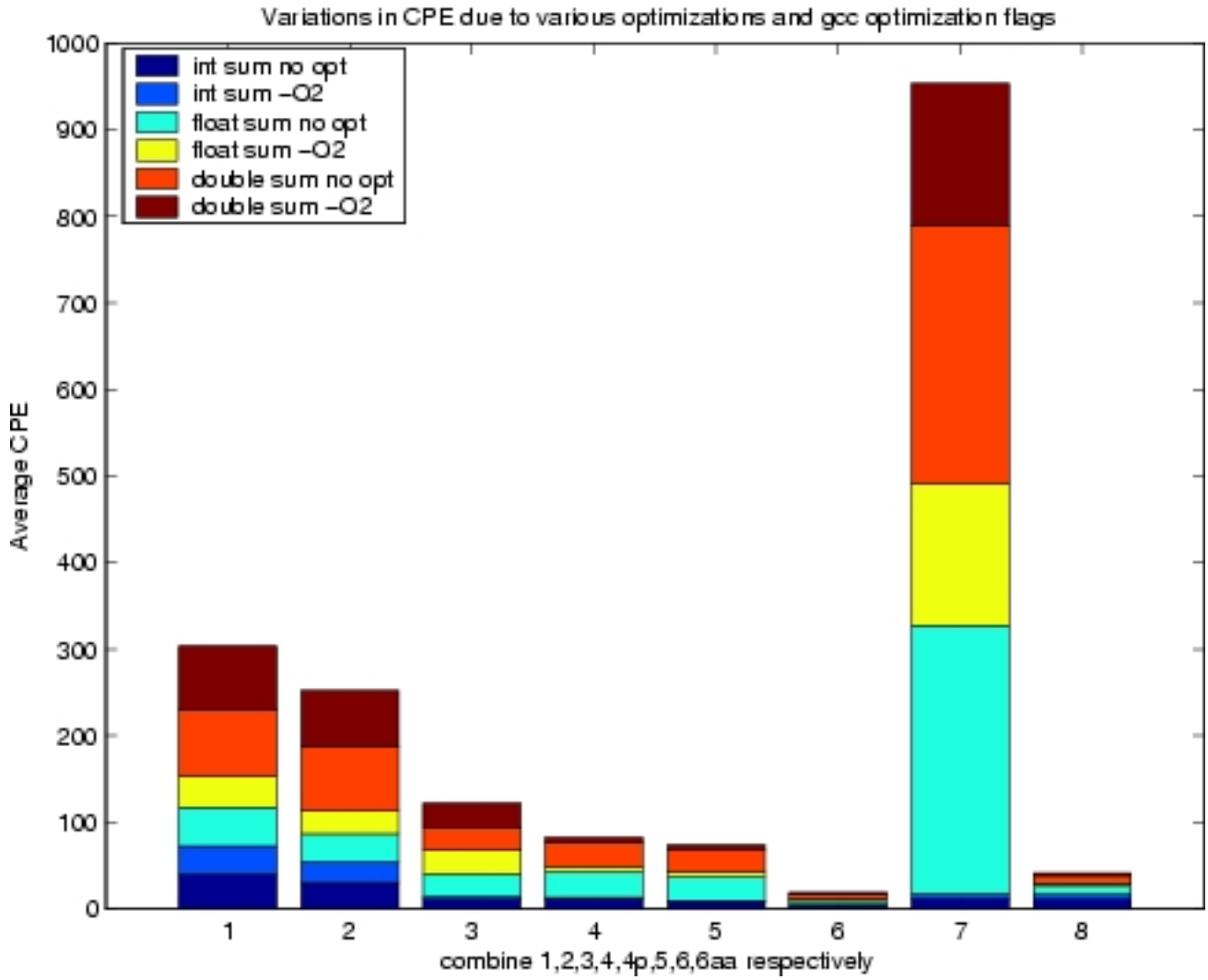


Figure 1: Here we see the differences between the various optimization techniques discussed in class. We note that the gcc -O2 flag doesn't come close to replacing good programming techniques. The code was run on a P4 2.4 GHz machine

```
for intel based machines, or

#ifdef NEW_CHUD
double count1,count2,count3,count4,count4p,count5,count6,count6aa,count6aaa;
double icount1,icount2,icount3,icount4,icount4p,icount5,icount6,icount6aa,icount6aaa;
#endif

for PPC based machines.

Then print out the value at the bottom of the routine
(again dependent on the architeture).

#ifdef X86_ON
printf("\ncycles=[%f,%f,%f,%f,%f,%f,%f,%f,%f];\n",count1,count2,
count3,count4,count4p,count5,count6,count6aa,count6aaa);
```

```
#endif

#ifdef NEW_CHUD
ount5,count6,count6aa,count6aaa);
printf("\ninstructions=[%f,%f,%f,%f,%f,%f,%f,%f,%f];\n",icount1,icount2,
icount3,icount4,icount4p,icount5,icount6,icount6aa,icount6aaa);
#endif
```

Finally include your file in the include.h file directly beneath the combine6aa.c. Setup the parameters.h and Makefile appropriately & enjoy!

In order to run the code for the G4/ Motorola 745x chip set one needs to have the CHUD development toolkit installed on their apple computer. Fortunately, the toolkit being used is the primary toolkit for performance analysis developed by Apple, and the GUI implies compatibility with the IBM G5 64 bit processors. Thus this toolkit should also work for these processors as well.
To choose between the G4 and x86 cycle counters, edit the parameters.h file and comment out or include the corresponding flag. From the parameters.h file showing the G4 cycle counter (NEW_CHUD) will be enabled, while the G4 GUI (MONster), and X86_ON flags are disabled.

```
/* Set this if you're running on an intel machine*/
//#define X86_ON

/*
I recommend using NEW_CHUD instead of CHUD_ON (which uses MONster)
I've found MONster doesn't take into account the latency of the PMC readers.
*/

#define NEW_CHUD
//#define CHUD_ON
```

MATLAB Vector output

Setting the NEW_CHUD or X86_ON flag will output to the screen a vector of values (Two for the NEW_CHUD flag one for the X86_ON flag. The output is in MATLAB vector format and scripts are available to plot the bar graphs shown throughout this paper. The cycles vector lists the number of clock cycles it took to perform the operations defined in call_combine.c. On the G4, there is an additional vector, instructions, that lists the number of instructions needed to perform the same operations.

MONster output

Running with the CHUD_ON flag will allow the remote access of the MONster program to read the registers being updated during your code. To setup MONster, you will need to do the following:

Launch MONster (/Developer/Applications/Performance Tools/CHUD/MONster.app)

select the Sampling Tab

choose User from the Privilege process filter pop up menu on the left

choose Marked from the Performance Mark filter pop up

choose 1- CPU Cycles from the PMC-1 event list on the right

choose Instruction from the PMC-2 event list on the right

press the Command+Shift+R keys to enable Remote Performance Monitoring Mode

select the Results Tab

Finally, run your code and MONster will write the results from your run in the Results window with Labels corresponding to the the name of the function being tested. MONster will output the results to a text file if you wish.

Availability

All the code, including the C combine routines, performance routines as well as the MATLAB plotting routines, makefiles, and pre-compiled binaries for both x86-linux and OS X-G4 platforms is available on my project website at:

`http://www.lcv.umd.edu/~palott/research/graduate/662/downloads/src`
and
`http://www.lcv.umd.edu/~palott/research/graduate/662/downloads/output`

CHUD is available from the macupdate website at:

`http://www.macupdate.com/info.php/id/8506`

Results

Apple G4/ Motorola 7455

From figure 2, we see that the results here are about 9 times slower that what we expected on the highly optimized combine5 code, which theoretically should obtain a CPE of 1.00, but instead we measure the CPE to be 8.9928. One would think that this is because the processor is burning up clock cycles to to cache misses, etc. especially since the G4 is suppose to be capable of performing 2 IPC (Instructions Per Cycle) for both integer and floating point arithmetic, and 1 IPC for doubles. Figure 3 shows us the number of instructions per clock cycle.
This shows us that even though we have optimized code, we don't necessarily get optimal results. It seems that through all of our optimizations we have made so few instructions that the clock is actually hungry for data. Thus we need a faster bus to feed the processor. It would be ideal if we could indeed prove this by measuring the number of instructions performed by the x86 machines with faster buses, or a G5 with a faster bus. Unfortunately we don't have access to information.

Pentium III and Pentium IV

From figure 4, we see that the results here are about what we expected on the highly optimized combine5 code, which theoretically should obtain a CPE of 1.00, but instead we measure the CPE to be 1.6915 on the 1.13 GHz PIII machine, and 1.4432 on the 2.4 GHz P4 machine. Oddly, however, both machines do much worse one the combine6 code that performs a product of the vector entries using a straight forward unrolling method. The error is corrected by doing a more clever parallel unrolling, but it is obvious that both machines are confused my the straightforward combine6 code for all data types except for integer. Even more strange is the fact that the code performs worse on the Pentium 4 machine.

```
void combine6(vec_ptr v, DATATYPE *dest)
{
  int i;
  int length=vec_length(v);
  int limit= length-1;
  DATATYPE *data=get_vec_start(v);
  DATATYPE x0=1;
  DATATYPE x1=1;
  DATATYPE sum=0;
/*
Combine 2 elements at a time
*/
  for (i=0;i<limit; i+=3) {
    x0*=data[i];
    x1*=data[i+1];
  }
  for(; i<length; i++){
     x0*=data[i];
  }
 *dest=x0+x1;
}
```

Summary/Conclusion

We have developed a suite of code that performs useful optimization analysis on several architectures, providing a simple way for programmers/scientists to analyze their code before investing in ”better” hardware. From our tests, we have shown that on average a 1.13 GHz P3 processor performs less clock cycles for the same operations than a 2.4GHz P4 processor. While a 1GHz G4 processor performs almost 10 times more clock cycles that a 1.13 GHz P3 processor, thus nullifying many claims of the G4 PPC processor. Testing one’s core code with such software can provide a useful measure of expectation before purchasing a new machine, or cluster to run ones code, and also serves as a development bed for testing new machine dependent optimization algorithms.

References

Computer Systems A Programmer’s Perspective. R. Bryant and D. O’Hallaron. Prentice Hall 2003.

Appendix

Raw Data

Note: To obtain the bar charts we used MATLAB's bar function with the following data.

G4

Clock Cycles

88.4828	39.2676	88.3124	39.4634	100.9066	50.4017
78.8806	37.6489	78.7765	38.0842	89.5357	48.9801
27.8623	13.2181	28.0469	13.9837	39.9443	26.0655
26.0360	9.0821	26.6462	10.1852	37.9702	20.3727
23.9951	9.7454	24.1947	10.7312	35.4855	21.3215
14.6235	8.9928	14.8139	10.3558	23.7701	20.6216
16.2511	8.9765	16.2889	10.1697	25.3217	20.3435
15.8821	8.6264	15.8860	9.6844	25.7340	20.3916
15.1290	9.4557	15.3702	10.2980	25.7842	20.3161

Instruction Count

58.0036	24.0022	58.0034	24.0023	58.0039	24.0024
48.0035	24.0022	48.0036	24.0020	48.0037	24.0027
16.0049	6.0023	16.0048	6.0021	16.0049	6.0028
14.0048	4.0023	14.0051	4.0024	14.0053	4.0028
11.0054	4.0029	11.0055	4.0028	11.0056	4.0033
7.7060	2.4026	7.7058	2.4023	7.7060	2.4032
7.6737	2.3369	7.6736	2.3368	7.6741	2.3374
7.0066	2.3366	7.0065	2.3365	7.0071	2.3370
8.1742	2.6698	8.1742	2.6697	8.1748	2.6700

Pentium

PIII Clock Cycles

41.9619	38.6834	43.3517	39.4787	57.0388	54.0593
34.7470	28.6405	34.7963	29.3651	56.5289	47.0554
10.4426	6.3208	10.6506	8.0614	31.5237	23.2627
8.8505	2.2265	9.4565	3.0526	13.0206	3.0295
8.1808	2.2177	8.0779	3.0478	14.4798	3.0345
4.1632	1.6915	4.5602	1.6829	5.8070	1.8671
4.6294	1.3491	42.6922	21.3690	47.0750	21.3865
4.3602	1.3505	4.4147	1.9468	10.1841	2.0109

PIV Clock Cycles

40.2740	32.4760	44.3892	35.8648	76.1108	74.4592
30.1948	24.1312	32.1856	26.3308	74.1976	65.2044
10.5876	3.4340	26.2012	27.1076	26.5232	28.0752
10.7632	2.0472	29.0892	6.0340	28.3756	6.0280
7.6048	2.0476	27.1300	5.0332	26.8936	5.0284
3.6772	1.4432	4.5968	2.1372	4.7372	2.2788
12.2144	4.8988	310.2800	162.9360	299.5640	163.6760
11.4840	5.1340	10.1264	2.3644	9.6504	2.3672

Vector Routines

Sums

combine1 - Uses a straight forward method to compute the vector sum. Accessing a non-local variable to update the sum during each iteration, as well as calling get_vec_element and vec_length during each iteration. Very inefficient. Poor use of locality.

combine2 - Same as combine1, but now computes and stores the vec_length on time outside the loop and access this local variable when needed.

combine3 - New routine called get_vec_start is called and then a local array index is updated to access the next vector entry each time. Otherwise the same as combine2.

combine4 - Same as combine3 except the sum is now stored in a local variable and copied to the non-local variable after the sum is computed.

combine4p - Same as combine4 except the pointer is accessed directly instead of through the array references. (actually generates worse assembly code).

combine5 - Same as combine4 except that we unroll the loop taking advantage of free integer or floating point operations.

Products

combine6 - Same as combine5, except we are now doing multiplication, and thus we attempt to do parallel unrolling. (Combining two elements at a time)

combine6aa - Same as combine6, except we multiply pairs of vector elements first, then multiply by the larger product, trying to reduce the ”tree height”.

combine6aaa - Same as combine6aa, except we multiply more pairs of vector elements first, then multiply by the larger product, trying to further reduce the ”tree height”.

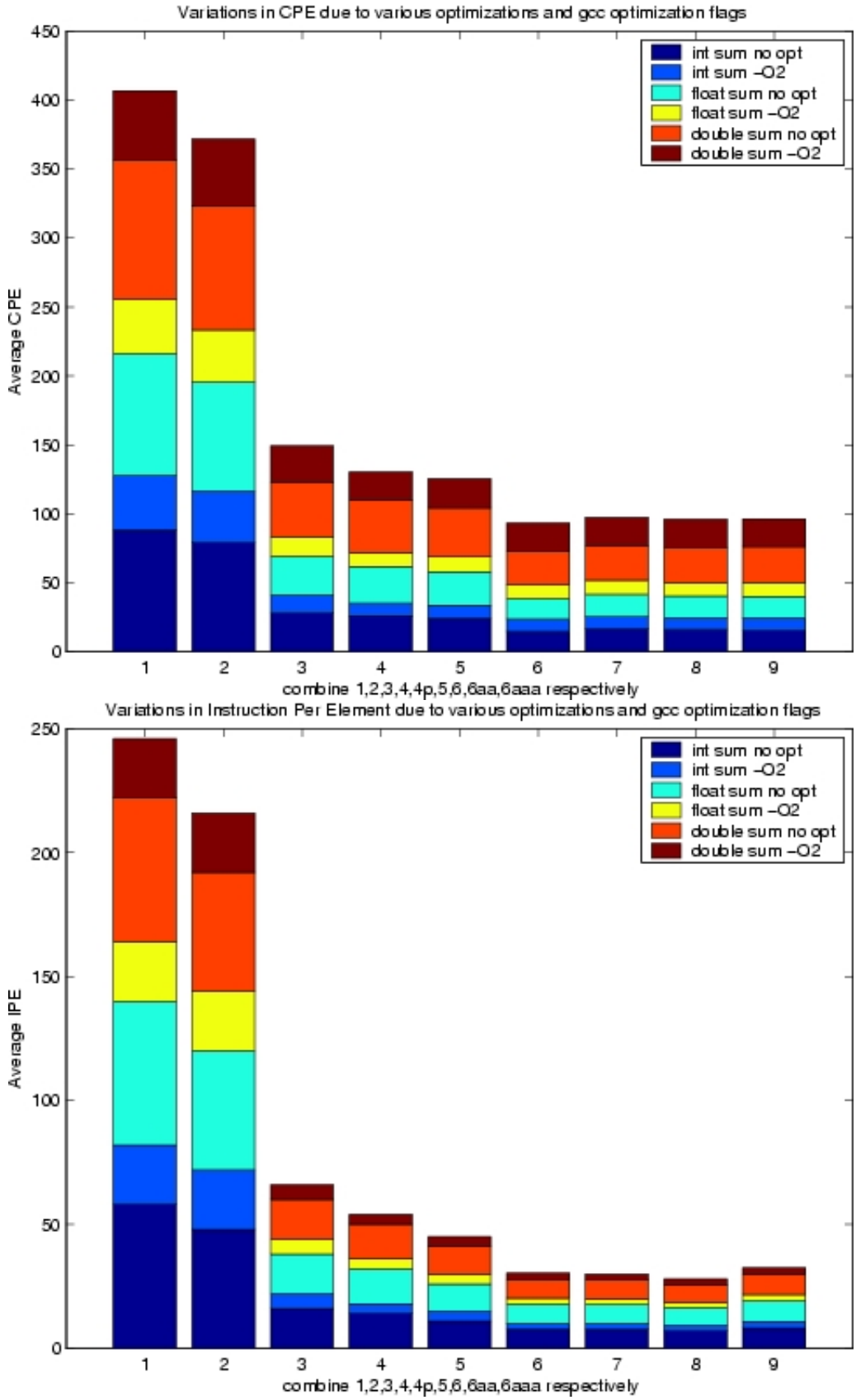


Figure 2: Performance Results from G4 Runs with 10000 elements. Cycles per element on top, Instructions per element on the bottom.

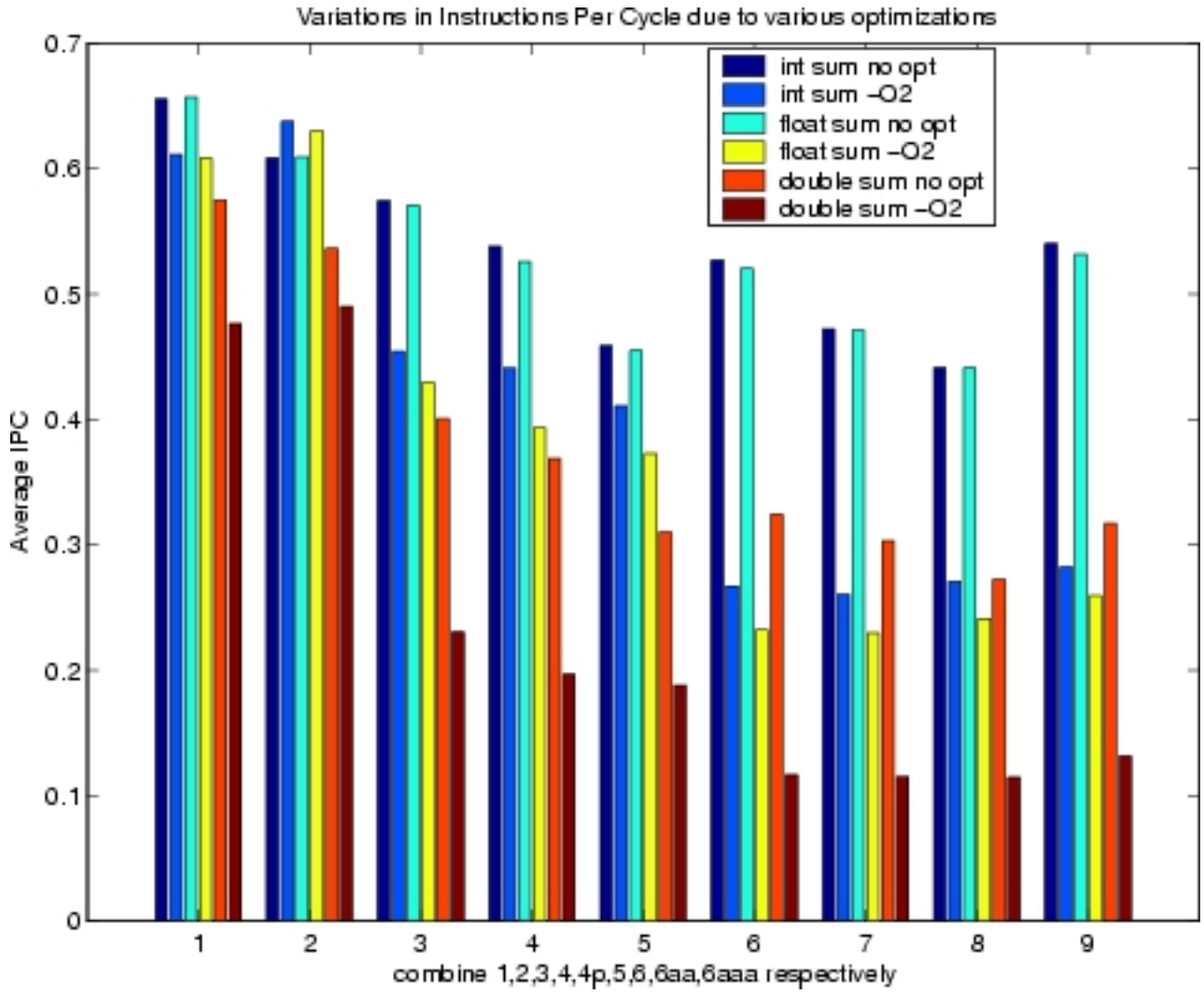
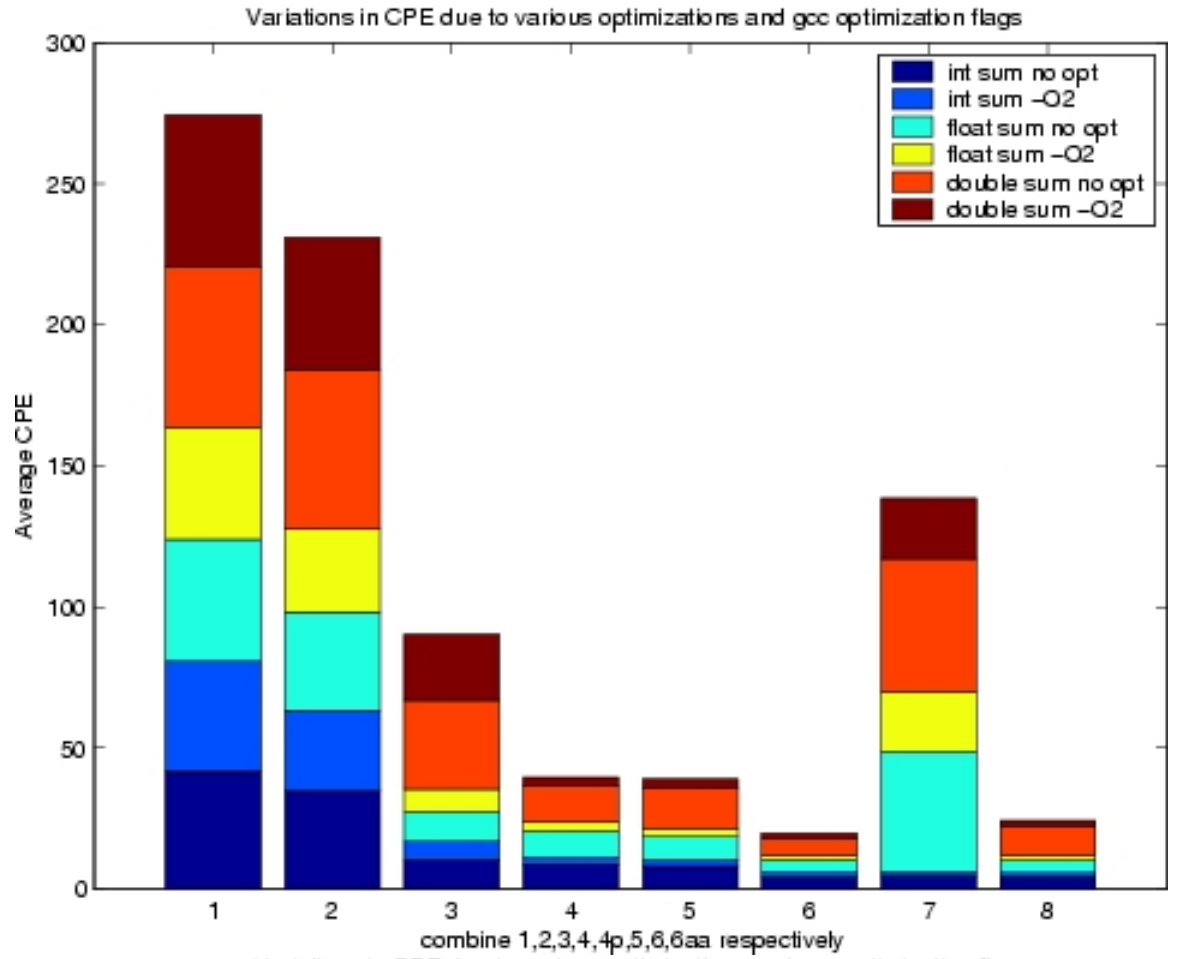


Figure 3: Performance Results from G4 Runs with 10000 elements. Instructions Per Cycle. Nowhere close to optimal performance



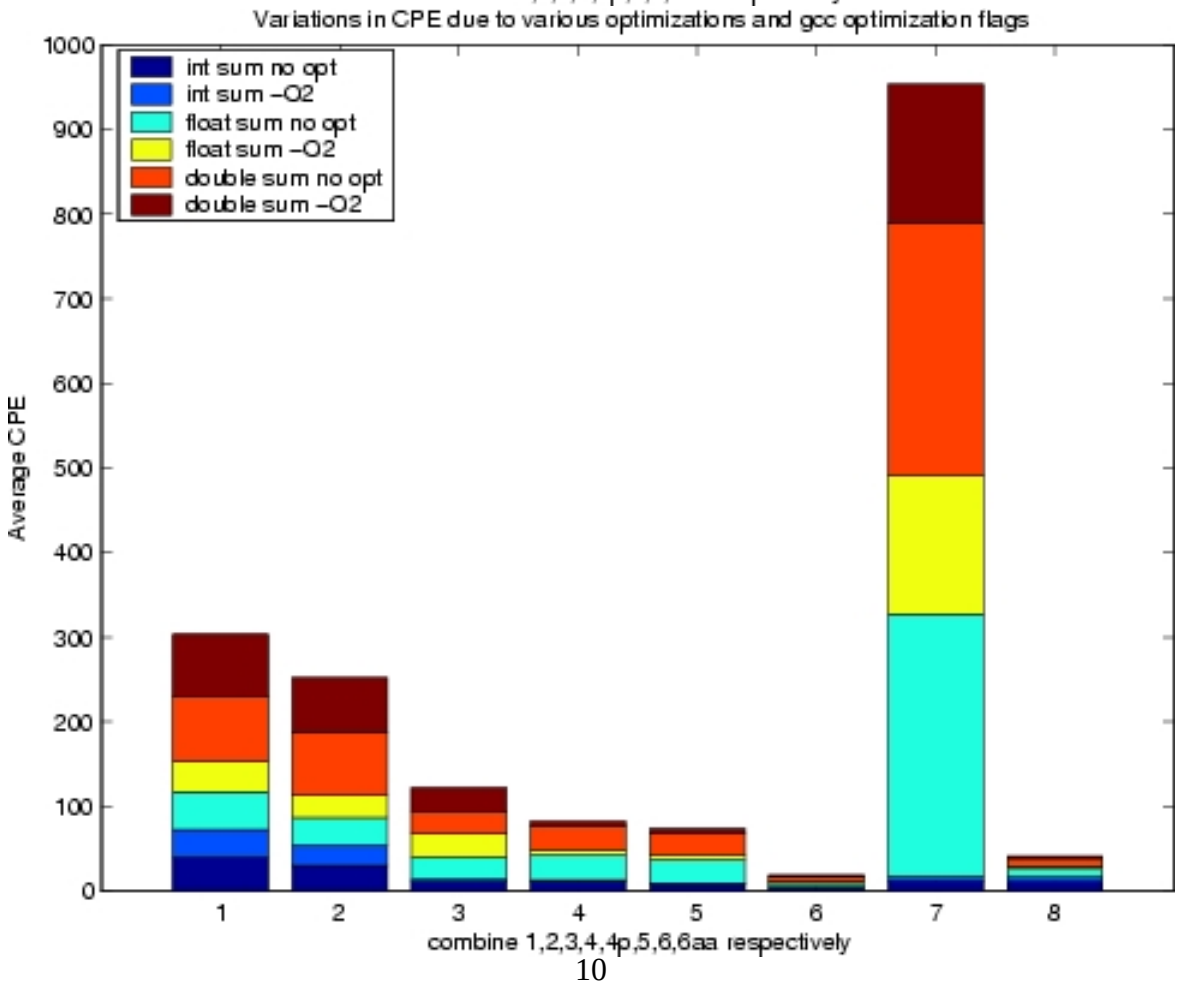


Figure 4: Performance Results from P3 and P4 Runs with 10000 elements respectively.